5

Classes and Objects

Key Concepts

- Using structures
- Creating a class
- Defining member functions
- Creating objects
- Using objects
- Inline member functions
- Nested member functions
- Private member functions
- Arrays as class members
- Storage of objects
- Static data members
- Static member functions
- Using arrays of objects
- Passing objects as parameters
- Making functions friendly to classes
- Functions returning objects
- **const** member functions
- Pointers to members
- Using dereferencing operators
- Local classes

5.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an

extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

5.2 C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
      char   name[20];
      int    roll_number;
      float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student. Example:

```
struct student A;  // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
        float x;
        float y;
};
        struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A;    // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

note

The only difference between a structure and a class in C++ is that, by default, the members of a class are *private*, while, by default, the members of a structure are *public*.

5.3 Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
        private:
             variable declarations;
             function declarations;
        public:
             variable declarations;
             function declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 5.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

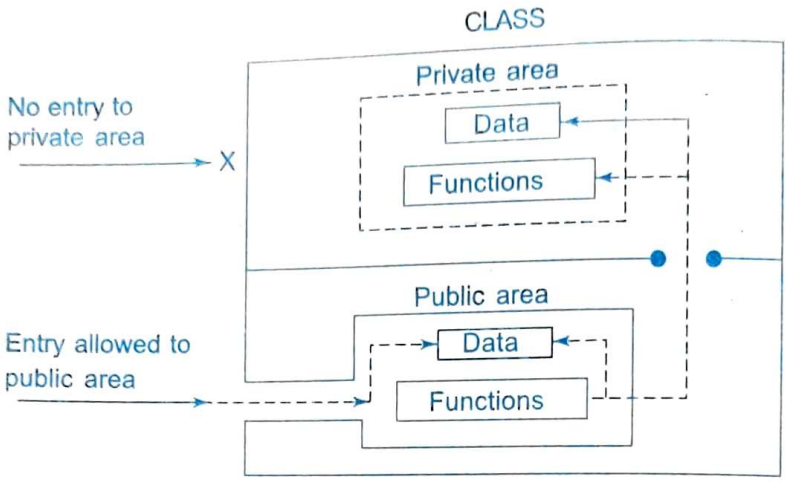

Fig. 5.1 ⇔ *Data hiding in classes*

A Simple Class Example

A typical class declaration would look like:

```
class item
{
        int number;                    // variables declaration
        float cost;                    // private by default
  public:
        void getdata(int a, float b);  // functions declaration
        void putdata(void);            // using prototype
  };// ends with semicolon
```

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 5.2 shows two different notations used by the OOP analysts to represent a class.

Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

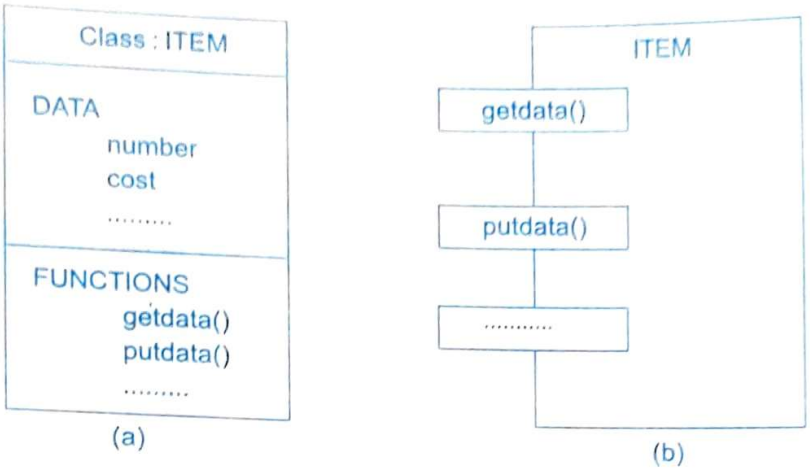

Fig. 5.2 ⇔ *Representation of a class*

```
item x;          // memory for x is created
```

creates a variable **x** of type **item**. In C++, the class variables are known as *objects*. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
        .....
        .....
        .....
} x,y,z;
```

would create the objects **x, y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function. Please refer Sec. 5.4 for further details.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

sends a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
      int x;
      int y;
  public:
      int z;
};
      .....
      .....
xyz p;
p.x = 0;          // error, x is private
p.z = 10          // OK, z is public
      .....
      .....
```

note

The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

5.4 Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the *ANSI prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to. The general form of a member function definition is:

```
return-type class-name :: function-name (argument declaration)
  {
        Function body
  }
```

The membership label class-name :: tells the compiler that the function *function-name* belongs to the class *class-name*. That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol :: is called the *scope resolution* operator.

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
     number = a;
     cost = b;
}
```

```
void item :: putdata(void)
{
    cout << "Number :" << number << "\n";
    cout << "Cost   :" << cost   << "\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are :

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
    int number;
    float cost;
  public:
    void getdata(int a, float b);        // declaration
        // inline function
    void putdata(void)                   // definition inside the class
    {
        cout << number  <<  "\n";
        cout << cost    <<  "\n";
    }
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an **inline** function are also applicable here. Normally, only small functions are defined inside the class definition.

5.5 A C++ Program with Class

All the details discussed so far are implemented in Program 5.1.

CLASS IMPLEMENTATION

```
#include <iostream>

using namespace std;

class item
{
      int number;    // private by default
      float cost;    // private by default
   public:
      void getdata(int a, float b);       // prototype declaration,
                                          // to be defined
      // Function defined inside class
      void putdata(void)
      {
            cout << "number :" << number << "\n";
            cout << "cost   :" << cost   << "\n";
      }
};
//............ Member Function Definition ..............
void item :: getdata(int a, float b)      // use membership label
{
      number = a;    // private variables
      cost = b;      // directly used
}
//..................... Main Program ....................

int main()
{
      item x;// create object x

      cout <<  "\nobject x "  <<  "\n";

      x.getdata(100, 299.95);             // call member function
      x.putdata();                        // call member function

      item y;                             // create another object

      cout <<  "\nobject y"  << "\n";

      y.getdata(200, 175.50);
      y.putdata();

      return 0;
  }
```

PROGRAM 5.1

This program features the class **item**. This class contains two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables. Note the use of statements such as

```
number = a;
```

in the function definition of **getdata()**. This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x, y;        // creates a list of objects
```

Here is the output of Program 5.1:

```
object x
number :100
cost   :299.95

object y
number :200
cost   :175.5
```

For the sake of illustration we have shown one member function as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.

5.6 Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
        .....
        .....
  public:
        void getdata(int a, float b);          // declaration
};
```

```
inline void item :: getdata(int a, float b)  // definition
{
      number = a;
      cost = b;
}
```

5.7 Nesting of Member Functions

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. Program 5.2 illustrates this feature.

NESTING OF MEMBER FUNCTIONS

```
#include <iostream>

using namespace std;

class set
{
      int m, n;
  public:
      void input(void);
      void display(void);
      int largest(void);
};

int set :: largest(void)
{
      if(m >= n)
            return(m);
      else
            return(n);
}

void set :: input(void)
{
      cout << "Input values of m and n" << "\n";
      cin >> m >> n;
}

void set :: display(void)
{
```

(Contd)

```
        cout << "Largest value = "
             << largest() << "\n";          // calling member function
}

int main()
{
        set A;
        A.input();
        A.display();

        return 0;
    }
```

PROGRAM 5.2

The output of Program 5.2 would be:

```
Input values of m and n
25 18
Largest value = 25
```

5.8 Private Member Functions

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
      int m;
      void read(void);              // private member function
  public:
      void update(void);
      void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();          // won't work; objects cannot access
                    // private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
    read();     // simple call; no object used
}
```

5.9 Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;      // provides value for array size

class array
{
    int a[size];        // 'a' is int type array
  public:
    void setval(void);
    void display(void);
};
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[ ]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 5.3 shows how these operations are implemented using a class with arrays as data members.

PROCESSING SHOPPING LIST

```
#include <iostream>

using namespace std;

const m=50;

class ITEMS
```

(Contd)